Introduction to OpenACC

Jeff Larkin

Some slides courtesy of John Levesque.

What is OpenACC?

- A common directive programming model for today's GPUs
  - Announced at SC11 conference
  - Offers portability between compilers
    - Drawn up by: NVIDIA, Cray, PGI, CAPS
    - Multiple compilers offer portability, debugging, permanence
  - Works for Fortran, C, C++
    - Standard available at www.OpenACC-standard.org
    - Initially implementations targeted at NVIDIA GPUs
- Current version: 1.0 (November 2011)
  - Version 2.0 RFC released at SC12
- Compiler support:
  - Cray CCE: nearly complete
  - PGI Accelerator: released product in 2012
  - CAPS: released product in Q1 2012

OpenACC Portability Goals

- Compiler Portability
  - Different compilers should support the same directives/ pragmas and runtime library
  - Work is currently underway to standardize a compliance test suite.
- Device Portability
  - Designed to be high level enough to support any of today's or tomorrow's accelerators.
  - Eliminate the need for separate code branches for CPU and GPUs.
- Performance Portability
  - Since OpenACC only annotated the code, well-written code should perform well on either the CPU or GPU

OPENACC BASICS

Important Directives to Know

- `!$acc parallel`
  - Much like `!$omp parallel`, defines a region where loop iterations may be run in parallel
  - Compiler has the freedom to decompose this however it believes is profitable
- `!$acc kernels`
  - Similar to parallel, but loops within the kernels region will be independent kernels, rather than one large kernel.
  - Independent kernels and associated data transfers may be overlapped with other kernels

Important Directives to Know

- `!$acc data`
  - Defines regions where data may be left on the device
  - Useful for reducing PCIe transfers by creating temporary arrays or leaving data on device until needed
- `!$acc host_data`
  - Define a region in which host (CPU) arrays will be used, unless specified with `use_device()`
  - Useful for overlapping with CPU computation or calling library routines that expect device memory

Important Directives to Know

- `!$acc wait`
  - Synchronize with asynchronous activities.
  - May declare specific conditions or wait on all outstanding requests
- `!$acc update`
  - Update a host or device array within a data region
  - Allows updating parts of arrays

Important Directives to Know

- `!$acc loop`
  - Useful for optimizing how the compiler treats specific loops.
  - May be used to specify the decomposition of the work
  - May be used to collapse loop nests for additional parallelism
  - May be used to declare kernels as independent of each other

Important Terminology

- Gang
  - The highest level of parallelism, equivalent to CUDA Threadblock. (num_gangs => number of threadblocks)
  - A “gang” loop affects the “CUDA Grid”
- Worker
  - A member of the gang, equivalent to CUDA thread within a threadblock (num_workers => threadblock size)
  - A “worker” loop affects the “CUDA Threadblock”
- Vector
  - Tightest level of SIMT/SIMD/Vector parallelism, equivalent to CUDA warp or SIMD vector length (vector_length should be a multiple of warp size)
  - A ‘vector” loop affects the SIMT parallelism

- Declaring these on particular loops in your loop nest will affect the decomposition of the problem to the hardware

USING OPENACC

Identify High-level, Rich Loop Nests

- Use your favorite profiling tool to identify hotspots at the highest level possible.
  - If there's not enough concurrency to warrant CUDA, there's not enough to warrant OpenACC either.

CrayPAT Loop-level profile

```
  100.0% | 117.646170 | 13549032.0 |Total
|-----------------------------------------------------
|  75.4% |  88.723495 | 13542013.0 |USER
||----------------------------------------------------
||  10.7% |  12.589734 |  2592000.0 |parabola_
|||---------------------------------------------------
3||   7.1% |   8.360290 |  1728000.0 |remap_.LOOPS
4||        |            |            | remap_
5||        |            |            |  ppmlr_
||||||-------------------------------------------------
6|||||   3.2% |   3.708452 |   768000.0 |sweepx2_.LOOP.2.li.35
7|||||        |            |            | sweepx2_.LOOP.1.li.34
8|||||        |            |            |  sweepx2_.LOOPS
9|||||        |            |            |   sweepx2_
10||||        |            |            |    vhone_
6|||||   3.1% |   3.663423 |   768000.0 |sweepx1_.LOOP.2.li.35
7|||||        |            |            | sweepx1_.LOOP.1.li.34
8|||||        |            |            |  sweepx1_.LOOPS
9|||||        |            |            |   sweepx1_
10||||        |            |            |    vhone_
||||||=================================================
3||   3.6% |   4.229443 |   864000.0 |ppmlr_
||||-------------------------------------------------
4|||   1.6% |   1.880874 |   384000.0 |sweepx2_.LOOP.2.li.35
5|||        |            |            | sweepx2_.LOOP.1.li.34
6|||        |            |            |  sweepx2_.LOOPS
7|||        |            |            |   sweepx2_
8|||        |            |            |    vhone_
4|||   1.6% |   1.852820 |   384000.0 |sweepx1_.LOOP.2.li.35
5|||        |            |            | sweepx1_.LOOP.1.li.34
6|||        |            |            |  sweepx1_.LOOPS
7|||        |            |            |   sweepx1_
8|||        |            |            |    vhone_
|||=================================================
```

Place OpenMP On High-level Loops

- Using OpenMP allows debugging issues of variable scoping, reductions, dependencies, etc. easily on the CPU
  - CPU toolset more mature
  - Can test anywhere
- Cray will soon be releasing Reveal, a product for scoping high-level loop structures.
- Who knows, this may actually speed-up your CPU code!

Focus on Vectorizing Low-Level Loops

- Although GPUs are not strictly vector processors, vector inner loops will benefit both CPUs and GPUs
  - Eliminate dependencies
  - Reduce striding
  - Remove invariant logic
  - …
- Compiler feedback is critical in this process

Finally, Add OpenACC

- Once High-Level parallelism with OpenMP and Low-Level vector parallelism is exposed and debugged, OpenACC is easy.

```
#ifdef _OPENACC
!$acc parallel loop private( k,j,i,n,r, p, e, q, u, v, w,&
!$acc&  svel0,xa, xa0, dx, dx0, dvol, f, flat,&
!$acc&  para,radius, theta, stheta) reduction(max:svel)
#else
!$omp parallel do private( k,j,i,n,r, p, e, q, u, v, w,&
!$omp&  svel0,xa, xa0, dx, dx0, dvol, f, flat,&
!$omp&  para,radius, theta, stheta) reduction(max:svel)
#endif
```

Differences between OpenMP and OpenACC

- Things that are different between OpenMP and OpenACC
  - Cannot have CRITICAL REGION down callchain
  - Cannot have THREADPRIVATE
  - Vectorization is much more important
  - Cache/Memory Optimization much more important
  - No EQUIVALENCE
  - Private variables not necessarily initialized to zero.
- Currently both OpenMP and OpenACC must be included in the source

```
#ifdef _OPENACC
!$acc parallel loop private( k,j,i,n,r, p, e, q, u, v, w,&
!$acc&  svel0,xa, xa0, dx, dx0, dvol, f, flat, para,radius,&
!$acc&  theta, stheta) reduction(max:svel)
#else
!$omp parallel do private( k,j,i,n,r, p, e, q, u, v, w, svel0,&
!$omp& xa, xa0, dx, dx0, dvol, f, flat, para,radius, &
!$omp& theta, stheta) reduction(max:svel)
#endif
```

Compiler list for SWEEPX1

```
45.                  #ifdef GPU
46.  G------------< !$acc parallel loop private( k,j,i,n,r, p, e, q, u, v, w, svel0,&
47.  G              !$acc&      xa, xa0, dx, dx0, dvol, f, flat, para,radius, theta, stheta)&
48.  G              !$acc&      reduction(max:svel)
49.  G              #else
50.  G              !$omp parallel do private( k,j,i,n,r, p, e, q, u, v, w, svel0,&
51.  G              !$omp&      xa, xa0, dx, dx0, dvol, f, flat, para,radius, theta, stheta)&
52.  G              !$omp&      reduction(max:svel)
53.  G              #endif
55.  G g----------< do k = 1, ks
56.  G g 3--------<  do j = 1, js
57.  G g 3               theta=0.0
58.  G g 3               stheta=0.0
59.  G g 3               radius=0.0
62.  G g 3 g------<    do i = 1,imax
63.  G g 3 g             n = i + 6
64.  G g 3 g             r  (n) = zro(i,j,k)
65.  G g 3 g             p  (n) = zpr(i,j,k)
66.  G g 3 g             u  (n) = zux(i,j,k)
67.  G g 3 g             v  (n) = zuy(i,j,k)
68.  G g 3 g             w  (n) = zuz(i,j,k)
69.  G g 3 g             f  (n) = zfl(i,j,k)
71.  G g 3 g             xa0(n) = zxa(i)
72.  G g 3 g             dx0(n) = zdx(i)
73.  G g 3 g             xa (n) = zxa(i)
74.  G g 3 g             dx (n) = zdx(i)
75.  G g 3 g             p  (n) = max(smallp,p(n))
76.  G g 3 g             e  (n) = p(n)/(r(n)*gamm)+0.5*(u(n)**2+v(n)**2+w(n)**2)
77.  G g 3 g------>    enddo
79.  G g 3             ! Do 1D hydro update using PPMLR
80.  G g 3 gr2 I--> call ppmlr (svel0, sweep, nmin, nmax, ngeom, nleft, nright,r, p, e, q, u, v, w, &
81.  G g 3                xa, xa0, dx, dx0, dvol, f, flat, para,radius, theta, stheta)
82.  G g 3
```

Compiler list for SWEEPX1

```
 ftn-6405 ftn: ACCEL File = sweepx1.f90, Line = 46
  A region starting at line 46 and ending at line 104 was placed on the accelerator.

ftn-6418 ftn: ACCEL File = sweepx1.f90, Line = 46
  If not already present: allocate memory and copy whole array "zro" to accelerator, free at line 104
(acc_copyin).

ftn-6418 ftn: ACCEL File = sweepx1.f90, Line = 46
  If not already present: allocate memory and copy whole array "zpr" to accelerator, free at line 104
(acc_copyin).

ftn-6418 ftn: ACCEL File = sweepx1.f90, Line = 46
  If not already present: allocate memory and copy whole array "zux" to accelerator, free at line 104
(acc_copyin).

ftn-6418 ftn: ACCEL File = sweepx1.f90, Line = 46
  If not already present: allocate memory and copy whole array "zuy" to accelerator, free at line 104
(acc_copyin).

ftn-6418 ftn: ACCEL File = sweepx1.f90, Line = 46
  If not already present: allocate memory and copy whole array "zuz" to accelerator, free at line 104
(acc_copyin).

ftn-6418 ftn: ACCEL File = sweepx1.f90, Line = 46
  If not already present: allocate memory and copy whole array "zfl" to accelerator, free at line 104
(acc_copyin).

ftn-6416 ftn: ACCEL File = sweepx1.f90, Line = 46
  If not already present: allocate memory and copy whole array "send1" to accelerator, copy back at line
104 (acc_copy).
```

But Now It Runs Slower!

- Every time I've gone through this process, the code is slower at this step than when I started.

- OpenACC is not automatic, you've still got work to do…
  - Improve data movement
  - Adjust loop decomposition
  - File bugs?

CrayPAT Profiling of OpenACC

- Craypat profiling
  - Tracing: "pat_build -u <executable>" (can do APA sampling first)
  - "pat_report -O accelerator <.xf file>"; -T also useful
    - Other pat_report tables (as of perftools/5.2.1.7534)
      - acc_fu flat table of accelerator events
      - acc_time call tree sorted by accelerator time
      - acc_time_fu flat table of accelerator events sorted by accelerator time
      - acc_show_by_ct regions and events by calltree sorted alphabetically

Run and gather runtime statistics

```
Table 1:  Profile by Function Group and Function

 Time % |      Time |Imb. Time |   Imb. | Calls |Group
        |           |          | Time % |       | Function
        |           |          |        |       |  PE='HIDE'
        |           |          |        |       |   Thread='HIDE'

 100.0% | 83.277477 |       -- |     -- | 851.0 |Total
|-----------------------------------------------------------------
|  51.3% | 42.762837 |       -- |     -- | 703.0 |ACCELERATOR
||----------------------------------------------------------------
||  18.8% | 15.672371 | 1.146276 |   7.3% |  20.0 |recolor_.SYNC_COPY@li.790←not good
||  16.3% | 13.585707 | 0.404190 |   3.1% |  20.0 |recolor_.SYNC_COPY@li.793←not good
||   7.5% |  6.216010 | 0.873830 |  13.1% |  20.0 |lbm3d2p_d_.ASYNC_KERNEL@li.116
||   1.6% |  1.337119 | 0.193826 |  13.5% |  20.0 |lbm3d2p_d_.ASYNC_KERNEL@li.119
||   1.6% |  1.322690 | 0.059387 |   4.6% |   1.0 |lbm3d2p_d_.ASYNC_COPY@li.100
||   1.0% |  0.857149 | 0.245369 |  23.7% |  20.0 |collisionb_.ASYNC_KERNEL@li.586
||   1.0% |  0.822911 | 0.172468 |  18.5% |  20.0 |lbm3d2p_d_.ASYNC_KERNEL@li.114
||   0.9% |  0.786618 | 0.386807 |  35.2% |  20.0 |injection_.ASYNC_KERNEL@li.1119
||   0.9% |  0.727451 | 0.221332 |  24.9% |  20.0 |lbm3d2p_d_.ASYNC_KERNEL@li.118
```

Optimizing Data Movement

- Compilers will be cautious with data movement a likely move more data that necessary.
  - If it's left of '=', it will probably be copied from the device.
  - If it's right of '=', it will probably be copied to the device.

- The CUDA Profiler can be used to measure data movement.
- The Cray Compiler also has the CRAY_ACC_DEBUG runtime environment variable, which will print useful information.
  - See `man intro_openacc` for details.

Optimizing Data Movement

- Step 1, place a data region around the simulation loop
  - Use this directive to declare data that needs to be copied in, copied out, or created resident on the device.
  - Use the present clause to declare places where the compiler may not realize the data is already on the device (within function calls, for example)
- Step 2, use an update directive to copy data between GPU and CPU inside the data region as necessary

Keep data on the accelerator with acc_data region

```
!$acc data copyin(cix,ci1,ci2,ci3,ci4,ci5,ci6,ci7,ci8,ci9,ci10,ci11,&
!$acc& ci12,ci13,ci14,r,b,uxyz,cell,rho,grad,index_max,index,&
!$acc& ciy,ciz,wet,np,streaming_sbuf1, &
!$acc&    streaming_sbuf1,streaming_sbuf2,streaming_sbuf4,streaming_sbuf5,&
!$acc&    streaming_sbuf7s,streaming_sbuf8s,streaming_sbuf9n,streaming_sbuf10s,&
!$acc&    streaming_sbuf11n,streaming_sbuf12n,streaming_sbuf13s,streaming_sbuf14n,&
!$acc&    streaming_sbuf7e,streaming_sbuf8w,streaming_sbuf9e,streaming_sbuf10e,&
!$acc&    streaming_sbuf11w,streaming_sbuf12e,streaming_sbuf13w,streaming_sbuf14w, &
!$acc&    streaming_rbuf1,streaming_rbuf2,streaming_rbuf4,streaming_rbuf5,&
!$acc&    streaming_rbuf7n,streaming_rbuf8n,streaming_rbuf9s,streaming_rbuf10n,&
!$acc&    streaming_rbuf11s,streaming_rbuf12s,streaming_rbuf13n,streaming_rbuf14s,&
!$acc&    streaming_rbuf7w,streaming_rbuf8e,streaming_rbuf9w,streaming_rbuf10w,&
!$acc&    streaming_rbuf11e,streaming_rbuf12w,streaming_rbuf13e,streaming_rbuf14e, &
!$acc&    send_e,send_w,send_n,send_s,recv_e,recv_w,recv_n,recv_s)
  do ii=1,ntimes
         o o o
      call set_boundary_macro_press2
      call set_boundary_micro_press
      call collisiona
      call collisionb
      call recolor
```

Now when we do communication we have to update the host

```
!$acc parallel_loop private(k,j,i)
  do j=0,local_ly-1
    do i=0,local_lx-1
      if (cell(i,j,0)==1) then
        grad (i,j,-1) = (1.0d0-wet)*db*press
      else
        grad (i,j,-1) = db*press
      end if
      grad (i,j,lz)   = grad(i,j,lz-1)
    end do
  end do
!$acc end parallel_loop
!$acc update host(grad)
  call mpi_barrier(mpi_comm_world,ierr)
  call grad_exchange
!$acc update device(grad)
```

But we would rather not send the entire grad array back – how about

Packing the buffers on the accelerator

```
!$acc data present(grad,recv_w,recv_e,send_e,send_w,recv_n,&
!$acc&                  recv_s,send_n,send_s)
!$acc parallel_loop
      do k=-1,lz
        do j=-1,local_ly
          send_e(j,k) = grad(local_lx-1,j        ,k)
          send_w(j,k) = grad(0          ,j        ,k)
        end do
      end do
!$acc end parallel_loop
!$acc update host(send_e,send_w)
      call mpi_irecv(recv_w, bufsize(2),mpi_double_precision,w_id, &
           tag(25),mpi_comm_world,irequest_in(25),ierr)
          o  o  o
      call mpi_isend(send_w, bufsize(2),mpi_double_precision,w_id, &
           tag(26),& mpi_comm_world,irequest_out(26),ierr)
      call mpi_waitall(2,irequest_in(25),istatus_req,ierr)
      call mpi_waitall(2,irequest_out(25),istatus_req,ierr)
!$acc update device(recv_e,recv_w)
!$acc parallel
!$acc loop
      do k=-1,lz
        do j=-1,local_ly
          grad(local_lx  ,j        ,k) = recv_e(j,k)
          grad(-1        ,j        ,k) = recv_w(j,k)
```

Final Profile - bulk of time in kernel execution

37.9%	236.592782	--	--	11403.0	ACCELERATOR
15.7%	98.021619	43.078137	31.0%	200.0	lbm3d2p_d_.ASYNC_KERNEL@li.129
3.7%	23.359080	2.072147	8.3%	200.0	lbm3d2p_d_.ASYNC_KERNEL@li.127
3.6%	22.326085	1.469419	6.3%	200.0	lbm3d2p_d_.ASYNC_KERNEL@li.132
3.0%	19.035232	1.464608	7.3%	200.0	collisionb_.ASYNC_KERNEL@li.599
2.6%	16.216648	3.505232	18.1%	200.0	lbm3d2p_d_.ASYNC_KERNEL@li.131
2.5%	15.401916	8.093716	35.0%	200.0	injection_.ASYNC_KERNEL@li.1116
1.9%	11.734026	4.488785	28.1%	200.0	recolor_.ASYNC_KERNEL@li.786
0.9%	5.530201	2.132243	28.3%	200.0	collisionb_.SYNC_COPY@li.593
0.8%	4.714995	0.518495	10.1%	200.0	collisionb_.SYNC_COPY@li.596
0.6%	3.738615	2.986891	45.1%	200.0	collisionb_.ASYNC_KERNEL@li.568
0.4%	2.656962	0.454093	14.8%	1.0	lbm3d2p_d_.ASYNC_COPY@li.100
0.4%	2.489231	2.409892	50.0%	200.0	streaming_exchange_.ASYNC_COPY@li.810
0.4%	2.487132	2.311190	48.9%	200.0	streaming_exchange_.ASYNC_COPY@li.625
0.2%	1.322791	0.510645	28.3%	200.0	streaming_exchange_.SYNC_COPY@li.622
0.2%	1.273771	0.288743	18.8%	200.0	streaming_exchange_.SYNC_COPY@li.574
0.2%	1.212260	0.298053	20.0%	200.0	streaming_exchange_.SYNC_COPY@li.759
0.2%	1.208250	0.422182	26.3%	200.0	streaming_exchange_.SYNC_COPY@li.806
0.1%	0.696120	0.442372	39.5%	200.0	streaming_exchange_.ASYNC_KERNEL@li.625
0.1%	0.624982	0.379697	38.4%	200.0	streaming_exchange_.ASYNC_KERNEL@li.525

Optimizing Kernels

- The compiler has freedom to schedule loops and kernels as it thinks is best, but the programmer can override this.
- First you must know how the work was decomposed.
  - Feedback from compiler at build time
  - Feedback from executable at runtime
  - CUDA Profiler

Adjusting Decomposition

- Adjust the number of gangs, workers, and or vector length on your `parallel` or `kernels` region
  - num_gangs, num_workers, vector_length
- Add `loop` directives to individual loop declaring them as gang, worker, or vector parallelism

Further Optimizing Kernels

- Use `loop collapse()` to merge loops and increase parallelism at particular levels
- Use compiler's existing directives regarding loop optimizations
  - Loop unrolling
  - Loop fusion/fission
  - Loop blocking
- Ensure appropriate data access patterns
  - Memory coalescing, bank conflicts, and striding are just as important with OpenACC as CUDA/OpenCL
  - This will likely help when using the CPU as well.

Interoperability

- OpenACC plays well with others; CUDA C, CUDA Fortran, Libraries
- If adding OpenACC to an existing CUDA code, the `deviceptr` data clause allows using existing data structures.
- If adding CUDA or a library call to an OpenACC code, use `host_data` and `use_device` to declare CPU or GPU memory use.

SHARING DATA BETWEEN OPENACC AND CUDA FOR C

OpenACC & CUDA C

- The Plan
  - Write a CUDA C Kernel and a Launcher function that accepts device pointers.
  - Write a C or Fortan main that uses OpenACC directives to manage device arrays
  - Use `acc host_data` pragma/directive to pass device pointer to launcher
  - Build .cu with nvcc and rest per usual
- Supported PEs: Cray, PGI

OpenACC C-main

- Notice that there is no need to create device pointers
- Use acc data region to allocate device arrays and handle data movement
- Use acc parallel loop to populate device array.
- Use acc host_data region to pass a device pointer for array

```
/* Allocate Array On Host */
  a = (double*)malloc(n*sizeof(double));

/* Allocate device array a. Copy data both to
   and from device. */
#pragma acc data copyout(a[0:n])
  {
#pragma acc parallel loop
    for(i=0; i<n; i++)
    {
      a[i] = i+1;
    }

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* Use device array when calling
   scaleit_launcher */
#pragma acc host_data use_device(a)
    {
      ierr = scaleit_launcher_(a, &n, &rank);
    }
  }
```

OpenACC Fortran-main

- Notice that there is no need to create device pointers
- Use acc data region to allocate device arrays and handle data movement
- Use acc parallel loop to populate device array.
- Use acc host_data region to pass a device pointer for array

```
integer,parameter :: n=16384
real(8) :: a(n)

!$acc data copy(a)
!$acc parallel loop
do i=1,n
  a(i) = i
enddo
!$acc end parallel loop

!$acc host_data use_device(a)
ierr = scaleit_launcher(a, n, rank)
!$acc end host_data
!$acc end data
```

SHARING DATA BETWEEN OPENACC AND LIBSCI

OpenACC and LibSCI

- The Plan:
  - Use OpenACC to manage your data
  - Possible use OpenACC for certain regions of the code
  - Use LibSCI's expert interface to call device routines
- Supported PEs: Cray

OpenACC with LibSCI - C

• OpenACC data region used to allocate device arrays for A, B, and C and copy data to/ from the device.

```
#pragma acc data copyin(a[0:lda*k],b
   [0:n*ldb]) copy(c[0:ldc*n])
   {
#pragma acc host_data use_device(a,b,c)
      {
          dgemm_acc
   ('n','n',m,n,k,alpha,a,lda,b,ldb,beta
   ,c,ldc);
      }
   }
```

OpenACC with LibSCI - Fortran

- OpenACC data region used to allocate device arrays for A, B, and C and copy data to/ from the device.

```
!$acc data copy(a,b,c)
!$acc host_data use_device(a,b,c)
      Call dgemm_acc
   ('n','n',m,n,k,alpha,a,lda,b,ldb,beta
   ,c,ldc)
!$acc end host_data
!$acc end data
```

Interoperability Advice

- OpenACC provides a very straightforward way to manage data structures without needing 2 pointers (host & device), so use it at the top level.
- CUDA provides very close-to-the-metal control, so it can be used for very highly tuned kernels that may be called from OpenACC
- Compilers do complex tasks such as reductions very well, so let them.